DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Juan Cruz Basso	627/14	jcbasso95@gmail.com
Brian Bohe	706/14	brianbohe@gmail.com
Francisco Figari	719/14	figafran@gmail.com
Ignacio Mariotti	651/14	mariotti.ignacio@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Este informe trata con una generalización de *traveling salesman problem* (TSP) en el que las ciudades proveen o requieren una cantidad conocida de un recurso y el vehículo que las recorre tiene capacidad máxima. En esta modificación del problema no es necesario pasar por todos los puntos que proveen el recurso, el punto de partida puede ser cualquier ciudad y las distancia entre las mismas es euclídea.

2. Descripción del Problema

Brian quiere ser un maestro Pokemon y para lograrlo debe vencer todos los gimnasios en su ciudad controlados por equipos enemigos. Un gimnasio tiene diferentes pokemones que deben ser derrotados para tomar control del mismo. Entre batallas de gimnasio se pueden utilizar pociones para curar a los pokemones. Las pociones pueden conseguirse en distintas poke paradas repartidas por la ciudad, cada poke parada provee 3 pociones y no se puede visitar dos veces.

Como Brian no tiene tiempo para mejorar sus pokemones decide ganar por fuerza bruta, esto implica conseguir suficientes pociones para tomar control de todos los gimnasios. Cada gimnasio requiere una cantidad de pociones conocida. La mochila de Brian tiene una capacidad limitada y no puede llevar todas las que quiera, por lo que debe planear bien su ruta. Las pociones que obtenga que sobrepasen la capacidad de la mochila serán descartadas.

Brian esta muy ocupado corrigiendo TPs, por lo que quiere saber cuánto tiempo le lleva vencer todos los gimnasios. El objetivo para convertirse en maestro Pokemon es vencer a todos los gimnasios recorriendo la mínima distancia, sin pasar por un gimnasio o una poke parada dos veces.

2.1. Consideraciones Previas

Para dar una descripción más formal del problema se establecen primero algunas notaciones. Sea un conjunto de gimnasios $G = \{g_1, g_2, \ldots, g_n\}$, poke paradas $P = \{p_1, p_2, \ldots, p_m\}$ y tamaño de mochila k la entrada para el problema. Sea *poc* una función que dado un gimnasio o una poke parada devuelve la cantidad de pociones que requiere o provee un nodo (valor negativo para los gimnasios y positivo para las paradas).

Se busca el camino mínimo $S = s_1, s_2, \ldots, s_l$ con s_i una poke parada o un gimnasio tal que $G \subseteq S$ y si k_i es la cantidad de pociones en la mochila luego de s_i se cumple que para todo $1 <= i <= l, k_i >= 0$. k_i se puede definir recursivamente como $k_i = min(k_{i-1} + poc(s_i), k)$. Se denota G_i y P_i a los conjuntos de poke paradas y gimnasios restantes luego de agregar s_i a S.

2.1.1. Observaciones

Si ningún gimnasio requiere pociones el problema se reduce a encontrar un camíno hamiltoniano mínimo entre todos ellos. Se deduce que no se va a poder aplicar un algoritmo polinomial para resolver el problema de manera exacta.

No todas las instancias del problema tienen solución y es dificil dar una condición suficiente y necesaria para garantizarla. Como condiciones necesarias se pueden definir:

- Tamaño de mochila chico: Si $k < max_{g \in G} poc(g)$, el tamaño de la mochila no es suficiente para ganar al menos uno de todos los gimnasios.
- Cantidad de pociones insuficientes: Si $\sum_{g \in G} poc(g) > \sum_{p \in P} poc(p)$. Es decir, la cantidad de pociones 'adquiribles' no son suficientes para vencer todos los gimnasios.

Una condición suficiente es:

- $\sum_{g \in G} poc(g) \leq \sum_{p \in P} poc(p)$ y $max_{g \in G} poc(g) + 2 \leq k$

Teniendo en cuenta que la cantidad de pociones proveídas en las poke paradas son mayores o igual a la cantidad requerida por gimnasios, solo podria no haber solucion en caso en que se tenga que desperdiciar pociones. Esta condición garantiza que no sea necesario descartar pociones para derrotar los gimnasios aprovechando que cada parada provee 3 pociones.

Sea $g' = maxarg_{g \in G} poc(g)$, dado una cantidad de pociones en la mochila en el paso i, k_i, existe q tal que $k_i + 3q \in \{ poc(g'), poc(g') + 1, poc(g') + 2 \}$. Dado que la mochila tiene una capacidad de $poc(g') + 2$ pociones, se puede garantizar que todos los gimnasios pueden ser vencidos sin desperdicio de pociones.

De todas las observaciones se desprende que luego de s_i no hay solución sii

$$(3 \cdot |P_i| + k_i) \leq \sum_{g \in G_i} poc(g) \tag{1}$$

Al valor de verdad de esta expresión se lo llama *SePuedeGanar*.

Para tener un mejor control sobre la experimentación todas las instancias del problema generadas cumplen la condición suficiente. Si se quiere ver más en detalle los resultados obtenidos en las experimentaciones se puede ingresar al repositorio en github[1].

[1] www.github.com/Gifaro/algo3_tp3

3. Algoritmo exacto

Un posible comienzo a la resolución del problema es la implementación de un algoritmo exacto. Se decidió utilizar backtracking partiendo independientemente desde cada uno de los puntos del mapa, ya sean paradas o gimnasios. Se denota G_i y P_i a los conjuntos de poke paradas y gimnasios restantes en el paso i de backtracking. Las podas implemetadas son las siguientes:

- Mochila negativa: Si la cantidad de pociones en la mochila se hace negativa, el camino es inválido.
- Pociones insuficientes: Si $\sum_{g \in G_i} poc(g) > 3 \cdot |P_i| + k_i$, es decir, si la cantidad de pociones requeridas en los gimnasios es mayor a las adquiribles mediante las paradas restantes más las guardadas en la mochila.
- Peor recorrido: Si el recorrido actual acumula una mayor distancia a la mejor solución encontrada hasta el momento.

3.1. Pseudocódigo y complejidad teórica

Algorithm 1 Backtracking

```
1:  procedure BACKTRACKING(P,G,k)
2:      S ← []                                              ▷ Secuencia vacia
3:      minima_distancia ← 0
4:      minimo_recorrido ← []
5:      for s ∈ P ∪ G do
6:          k_aux ← MIN(k_actual + poc(s),k)                ▷ k es la capacidad de la mochila
7:          BACKTRACKING_RECURSION(P-{s},G-{s},S+[s],k_aux)
8:      end for
9:      return minima_distancia, minimo_recorrido
10: end procedure
```

Algorithm 2 Recursión Backtracking

```
1: procedure BACKTRACKING_RECURSION(P,G,S,k_actual)
2:     if k_actual < 0 then
3:         return
4:     end if
5:     if 3 · |P| + k_actual < ∑_{g∈G} poc(g) then
6:         return
7:     end if
8:     if DISTANCIA(S)≥ minima_distancia then
9:         return
10:    end if
11:    if VACIO(G) then
12:        if DISTANCIA(S)< minima_distancia then
13:            minima_distancia ← DISTANCIA(S)
14:            minimo_recorrido ← S
15:        end if
16:    end if
17:    for s ∈ P ∪ G do
18:        k_aux ← MIN(k_actual + poc(s),k)          ▷ k es la capacidad de la mochila
19:        BACKTRACKING_RECURSION(P-{s},G-{s},S+[s],k_aux)
20:    end for
21: end procedure
```

Cada llamada recursiva achica el conjunto de nodos a visitar $P \cup G$ en 1 y corre $|P \cup G|$ recursiones. La recursión sin podas termina cuando $G = \emptyset$, como peor caso termina cuando $P \cup G = \emptyset$.

Se arma un árbol de recursión tal que en la base se realizan $m + n$ llamadas recursivas, en el próximo nivel $m + n - 1$, en el tercero $m + n - 2$ y así sucesivamente. Si en la base del árbol se actualiza el recorrido mínimo obtenido se realiza una copia del mismo, que en peor caso resulta $\mathcal{O}(m + n)$.

La cantidad de operaciones entonces se da por la expresión $(m+n) \cdot \prod_{i=0}^{m+n} i = (m+n) \cdot (m+n)! = \mathcal{O}((m + n + 1)!)$.

3.2. Experimentación

Se evaluó el algoritmo para 100 instancias diferentes para $(m + n) \in \{5, 6, \cdots, 13\}$ con diferentes relaciones de gimnasios y paradas. No se pudieron resolver instancias de más de 16 nodos. En la figura 3.1 se puede ver que los peores casos ocurren cuando la densidad de gimnasios es mayor. Incrementar las poke paradas para un gimnasio fijo resultó en un menor aumento del tiempo de cómputo que al realizar lo análogo para los gimnasios (aumentarlos fijando una poke parada).

También se evaluaron las distintas condiciones de corte (ver figura 3.2). Se puede ver que en general las instancias con una densidad de gimnasios menor tiene menos podas. Se puede conjeturar que las podas ocurren en un nivel más alto en el arbol de recursión, por lo que se podan más ramas y disminuye el tiempo de ejecución.

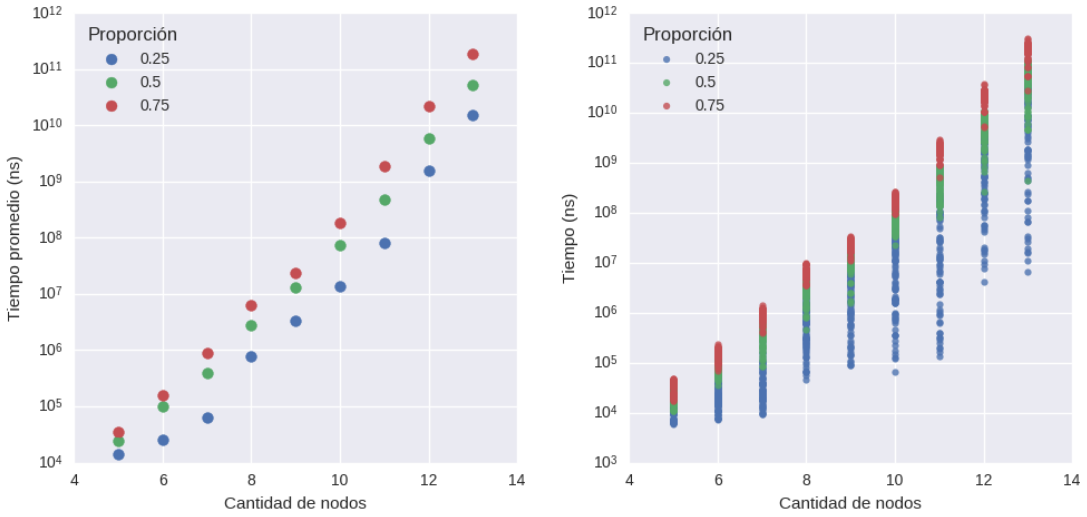


Figura 3.1: Tiempo de cómputo en base a la cantida de nodos de instancia, ratio indica la relación gimnasios/poke paradas

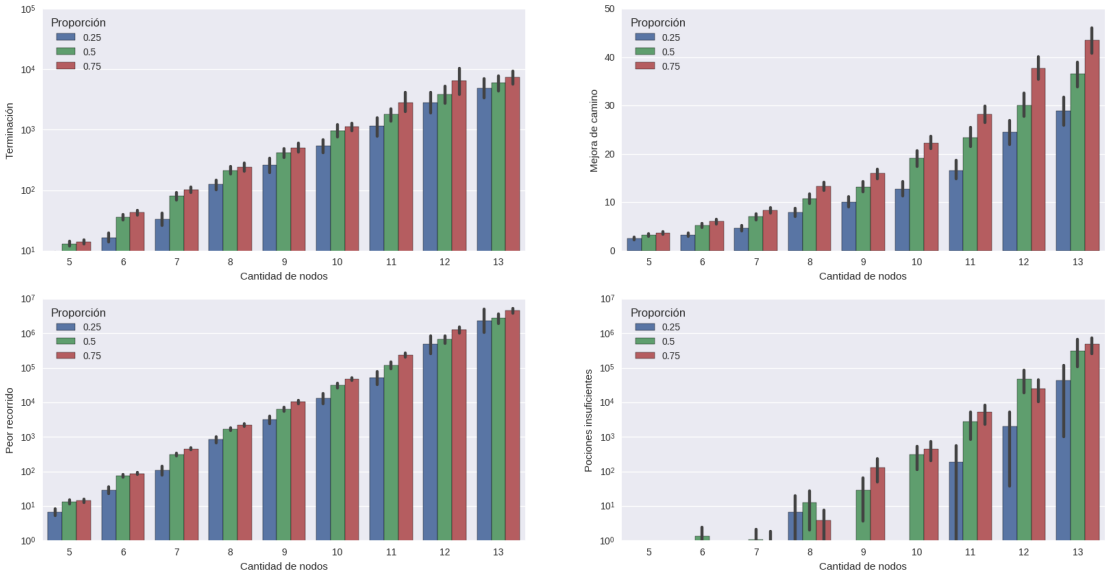


Figura 3.2: Promedio de podas y cortes de diferentes instancias

4. Heurística golosa

En esta sección se plantean dos heurísticas golosas con una complejidad temporal menor a la solución propuesta previamente pero que no obtienen necesariamente un resultado óptimo. Se proponen mejoras y adaptaciones para aleatorizar las soluciones obtenidas y se experimenta con ambas implementaciones buscando obtener los mejores resultados (menores distancias). Se asume la notación utilizada en la introducción.

4.1. Más Cercano

Esta heurística se basa en comenzar el recorrido con una parada al azar y armar luego progresivamente el camino, tomando el gimnasio más cetrcano, y si no lo fuera tomando la parada más cercana. Se definen $\hat{g} = minarg_{g \in G_i}\ Distancia(s_{i-1}, g)$ y $\hat{p} = minarg_{p \in P_i}\ Distancia(s_{i-1}, p)$ tomando distancia euclídea, respectivamente el gimnasio y la parada más cercanos al último punto del recorrido. De esta manera, si s_1 es el punto de partida y s_{i-1} el último punto del recorrido, se toma $s_i = \hat{g}$ si hay algun gimnasio no visitado y se tienen suficientes pociones para ir a él. Caso contrario se toma $s_i = \hat{p}_i$

El algoritmo termina si luego de agregar al i-ésimo elemento $G_i = \emptyset \vee \neg SePuedeGanar$.

Observación. *Dependiendo del orden en que se seleccionaron los diferentes puntos, puede no llegarse a una solución aún si esta existe. Asumiendo las condiciones suficientes descritas en la introducción*[2] *se asegura que las heurísticas siempre encuentran una solución.*

4.1.1. Pseudocódigo

```
1:  procedure MasCercano(P,G,k)
2:      S ← TomarAlAzar(P)
3:      P ← P - TomarAlAzar(P)
4:      while ¬ Vacio(G)∧ SePuedeGanar  do
5:          ĝ ← DameMasCercano(G, Ultimo(S))
6:          if 0 ≤ poc(ĝ)+ k_|S| then
7:              distancia ← distancia + calcularDistancia(Ultimo(S), ĝ)
8:              S ← AgregarGimnasio(S, ĝ)
9:              G ← G - ĝ
10:         else
11:             p̂ ← DameMasCercano(P, Ultimo(S))
12:             distancia ← distancia + calcularDistancia(Ultimo(S), p̂)
13:             S ← AgregarPokeParada(S, p̂)
14:             P ← P - p̂
15:         end if
16:     end while
17:     if ¬ Vacio(G) then
18:         distancia ← -1
19:     end if
20: end procedure
```

Donde

[2] Ver página 4

- k_i y las funciones POC y SEPUEDEGANAR corresponden a la notación descrita en la introducción.
- Las funciones AGREGARGIMNASIO y AGREGARPOKEPARADA agregan un gimnasio o una poke parada respectivamente al final de una secuencia, todos parámetros de la función, y definen el próximo valor de la sucesión de valores k_x.
- ULTIMO devuelve el último elemento de la secuencia pasada por parámetros.
- TOMARALAZAR toma una secuencia P y devuelve el r-ésimo elemento, con $r \in [0, |P|)$ generado de manera aleatoria.
- CALCULARDISTANCIA toma dos puntos y devuelve la distancia euclídea entre ellos.
- S es el camino de longitud DISTANCIA resultado de evaluar la heurística sobre cierta entrada.

```
1: procedure DameMasCercano(L, p)
2:     min_dist ← 0
3:     for p̂ ∈ L do
4:         if min_dist = 0 ∨ CalcularDistancia(p, p̂)≤ min_dist then
5:             min_dist = CalcularDistancia(p, p̂)
6:             res ← p̂
7:         end if
8:     end for
9:     return res
10: end procedure
```

Se itera $|L|$ veces realizando operaciones $\mathcal{O}(1)$, por lo que se deduce que DAMEMASCERCANO tiene una complejidad temporal $\mathcal{O}(|L|)$.

4.1.2. Complejidad Teórica

En la i-ésima iteración del algoritmo sólo se hace referencia al valor k_{i-1} como $\sum_{g \in G_{i-1}} \text{POC}(g)$, por lo que se los persistió en dos variables. También se representó cada gimnasio y poke parada con una estructura que consiste en su posición y su respectiva imagen en la función POC. Esto nos permite acceder al valor k_i y realizar llamadas a las funciones POC y SEPUEDEGANAR con una complejidad temporal $\mathcal{O}(1)$.

Implementando P, G, y S con listas se puede asumir las operaciones básicas como quitar o agregar un elemento, acceder al último valor y su tamaño en $\mathcal{O}(1)$. Por lo que las funciones VACIO, ULTIMO, AGREGARGIMNASIO y AGREGARPOKEPARADA son $\mathcal{O}(1)$.

Asumiendo que se puede generar un número aleatorio en $\mathcal{O}(1)$, TOMARALAZAR a lo sumo recorre el arreglo linealmente para acceder a una posición específica, lo cual es $\mathcal{O}(|P|)$ con P parámetro de entrada de la función utilizando listas.

En cada iteración se agrega una poke parada o un gimnasio a S incrementando así su tamaño en uno. Por definición del problema S puede ser tan grande como $P \cup G$ por lo que a lo sumo se itera tantas veces como $|P \cup G| = |P| + |G| = m + n$, con $n = |G|$ y $m = |P|$. Notar que sólo se realiza un llamado a DAMEMASCERCANO con P si en esa iteración se agrega una poke parada. Entonces hay n iteraciones que realizan $\mathcal{O}(n)$ operaciones y m con $\mathcal{O}(n+m)$. Se deduce que MASCERCANO tiene una complejidad temporal $\mathcal{O}(n^2 + m^2 + n \cdot m)$.

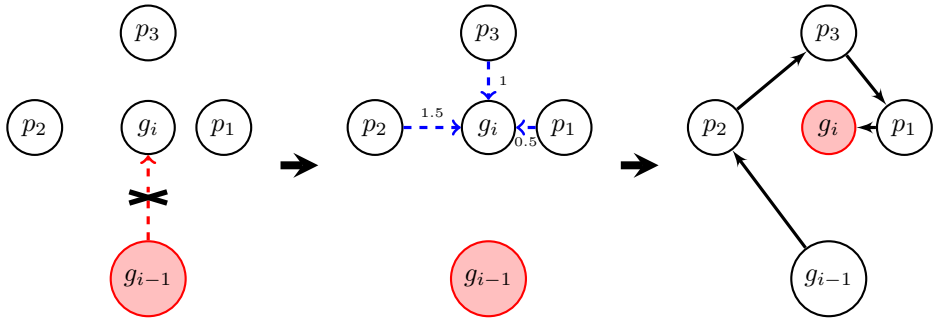


Figura 4.1: Ejemplo con $\text{POC}(g_i) + k_{i-1} < 0$ y $t = 3$. El nodo rojo corresponde al último lugar visitado por S.

4.2. Menos Pociones

Esta heurística se basa en recorrer los gimnasio de menor a mayor en base a la cantidad de pociones que estos requieran para ser vencidos, $|\text{POC}(g)|$. Respetando la notación utilizada se traduce a que el gimnacio g_1 se recorre antes que g_2 si y sólo si $\text{POC}(g_2) \leq \text{POC}(g_1)$, ya que $(\forall g \in G)\ \text{POC}(g) \leq 0$.

En el caso de no contar con suficientes pociones en la mochila para ganarle al gimnasio siguiente g_i como se ve en la figura 4.1, se agregan a S las t poke paradas más cercanas a g_i, con $t = \lceil \frac{|k_{i-1}+\text{POC}(g_i)|}{3} \rceil \leq \lceil \frac{|\text{POC}(g_i)|}{3} \rceil$, ordenadas de mayor a menor distancia euclídea respecto a g_i.

4.2.1. Pseudocódigo

Algorithm 3 MenosPociones

```
1:  procedure MenosPociones(P, G, k)
2:      Sort(G)
3:      while ¬ Vacio(G)∧ SePuedeGanar() do
4:          ĝ ← DameMasDebil(G)
5:          while 0 ≤ poc(ĝ)+ k_|S| ∧ ¬ Vacia(P) do
6:              p̂ ← DameMasCercano(ĝ, P)
7:              tramo ← AgregarPokeParada(tramo, p̂)
8:              P ← P−p̂
9:          end while
10:         if 0 ≤ poc(ĝ)+ k_|S| then
11:             S ← AgregarGimnasio(tramo, ĝ)
12:             G ← G−ĝ
13:             distancia ← distancia + UnirCaminos(S,tramo)
14:         end if
15:     end while
16:     if ¬ Vacio(G) then
17:         distancia ← - 1
18:     end if
19: end procedure
```

Donde

- Las funciones POC, ULTIMO, DAMEMASCERCANO, AGREGARGIMNASIO y AGREGARPOKEPARADA y los valores k_i son análogos a la heurística *Mas Cercano*[3].
- El SORT aplicado ordena de la forma forma descrita previamente.
- DAMEMASDEBIL toma el primer elemento de la secuencia ya que se encuentra ordenado.
- S es el camino de longitud DISTANCIA resultado de evaluar con la heurística cierta entrada.

Algorithm 4 Unir Caminos

```
1: procedure UnirCaminos(S, L)
2:     res ← 0
3:     if Vacio(S) then
4:         S ← Primero(L)
5:         QuitarPrimero(L)
6:     end if
7:     while ¬ Vacio(L) do
8:         res ← res + CalcularDistancia(Ultimo(S),Primero(L))
9:         S ← Primero(L)
10:        QuitarPrimero(L)
11:    end while
12:    return res
13: end procedure
```

Se itera $|L|$ veces realizando operaciones $\mathcal{O}(1)$, por lo que se deduce que UNIRCAMINOS tiene una complejidad temporal $\mathcal{O}(|L|)$.

4.2.2. Complejidad Teórica

Se consideraron las mismas variables implementadas de manera análoga a la heurística anterior, consiguiendo acceso a los valores k_i y llamadas a las funciones POC, QUITARPRIMERO y PRIMERO en $\mathcal{O}(1)$.

Sin considerar el ciclo más chico se puede ver que en cada iteración se incrementa en uno la cantidad de gimnasio en S y el algoritmo termina cuando $G \subseteq S$ o $\neg SePuedeGanar$, por lo que a lo sumo hay $|G|$ iteraciones.

Dentro de cada iteración se encuentran dos ciclos los cuales uno realiza sólo operaciones $\mathcal{O}(1)$ y el otro una busqueda lineal en P. Ambos se iteran tantas veces como poke paradas sean necesarias agregar para ganarle al gimnasio, las cuales para un gimnasio g son $\text{POC}(g)$. Entonces en el total de las $|G|$ iteraciones principales se necesitan agregar a lo sumo $\hat{p} = \sum_{g \in G} \frac{\text{POC}(g)}{3}$ poke paradas, realizando a lo sumo $\hat{p} \cdot |P|$ operaciones en total.

Se deduce entonces que la heurística *MenosPociones* tiene una complejidad temporal $\mathcal{O}(|G| \cdot log(|G|) + \hat{p} \cdot |P|) = \mathcal{O}(n \cdot log\, n + \hat{p} \cdot m)$.

4.3. Mejora "Más Cercano"

Se propuso una modificación como posible mejora al primer algoritmo. Suponiendo que se está por agregar al gimnasio g como el elemento s_{i+1} en el camino (con $1 < i$), la idea consiste en tomar estrategicamente una poke parada $p \in P_i$ y agregarla antes que g. Eligiendo $p =$

[3] Ver página 9

$argmin_{p \in P_i}$ $distancia(s_i, p) + distancia(p, g) \leq distancia(s_i, g) \cdot (1 + \gamma)$ como se ve en la figura 4.2, con $\gamma \in \mathbf{N}_{\geq 0}$ nuevo parámetro del algoritmo. De no existir p que cumpla la condición, no se agregan elementos antes que g. Intuitivamente la idea se basa en pagar un costo bajo en distancia al desviarse de camino a un gimnasio para evitar luego estar obligado a moverse a una poke parada por no tener suficientes pociones.

Se definieron condiciones necesarias para realizar el desvío:

- $P_i \neq \emptyset$: hay poke paradas que no se visitaron.
- $k_i < k$: la mochila no se llenó, no tiene sentido desperdiciar pociones.
- $\text{POC(g)} \leq \text{MIN}(k_i + 3, k)$: no tiene sentido desviarse si de todas formas no se le puede ganar al gimnasio.
- $\sum_{g \in G_i} \text{POC}(g) \leq 3 \cdot (|P_i| - 1) + \text{MIN}(k_i + 3, k)$: si se desperdician 1 o 2 pociones al tomar el desvío, sigue valiendo *SePuedeGanar*.

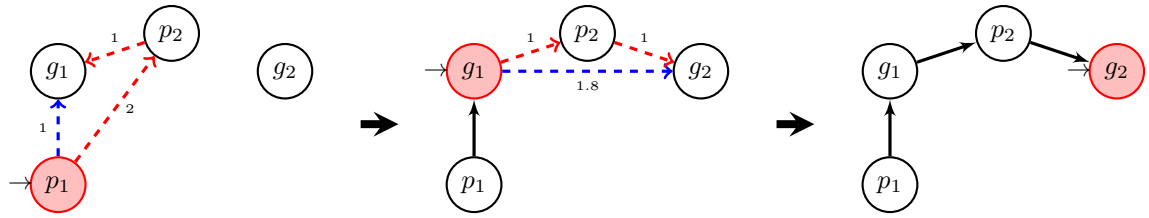


Figura 4.2: Ejemplo con $\gamma = 0{,}15$. Señalizado con una flecha y en rojo el último elemento de S.

4.3.1. Pseudocódigo

Implementar esta mejora significó el agregado de una función y cambios en el pseudocódigo de *MasCercano*.

Algorithm 5 Mejora "Más Cercano"

```
1: procedure MasCercano(P, G, k, γ)
2:     S ← TomarAlAzar(P)
3:     P ← P - TomarAlAzar(P)
4:     while ¬ Vacio(G)∧ SePuedeGanar() do
5:         ĝ ← DameMasCercano(G, Ultimo(S))
6:         intermedio ← DameElementoIntermedio(Ultimo S, ĝ, P, γ)
7:         if 0 ≤ poc(ĝ)+ k_|S| ∨ Valido(intermedio) then
8:             if Valido(intermedio) then
9:                 distancia ← distancia + calcularDistancia(Ultimo(S), intermedio)
10:                S ← AgregarPokeParada(S, intermedio)
11:                P ← P - intermedio
12:            end if
13:            distancia ← distancia + calcularDistancia(Ultimo(S), ĝ)
14:            S ← AgregarGimnasio(S, ĝ)
15:            G ← G - ĝ
16:        else
17:            p̂ ← DameMasCercano(P, Ultimo(S))
18:            distancia ← distancia + calcularDistancia(Ultimo(S), p̂)
19:            S ← AgregarPokeParada(S, p̂)
20:            P ← P - p̂
21:        end if
22:    end while
23:    if ¬ Vacio(G) then
24:        distancia ← -1
25:    end if
26: end procedure
```

Algorithm 6 DameElementoIntermedio

```
1: procedure DameElementoIntermedio(s, g, L, γ)
2:     min_dist ← 0
3:     if HayCondicionesNecesarias() then
4:         dist ← CalcularDistancia(s, g)
5:         dist ← dist · (1 + γ)
6:         for p ∈ L do
7:             if CalcularDistancia(s, p)+ CalcularDistancia(p, g)≤ dist then
8:                 dist = CalcularDistancia(s, p)+ CalcularDistancia(p, g)
9:                 res ← p
10:            end if
11:        end for
12:    end if
13:    return res
14: end procedure
```

Se puede ver fácilmente que la complejidad temporal de DameElementoIntermedio es $\mathcal{O}(|L|)$ con L parámetro de la función ya que se itera $|L|$ veces sobre un bloque de operaciones $\mathcal{O}(1)$.

Sólo de modifico el pseudocódigo de *MasCercano* anterior agregandodo al comienzo de cada iteración un llamado a la función DameElementoIntermedio con parámetro P. Se deduce entonces que la nueva complejidad temporal de *MasCercano* o de *MejoraMasCercano* es $\mathcal{O}((n+m)\cdot(n+m)) = \mathcal{O}(n^2+m^2+m\cdot n)$.

4.4. Aleatorizando las implementaciones

Se planteó introducir una parámetro $\alpha \in [0,1]$ a las heurísticas descritas previamente que permitiese variar el camino S resultante.

Se estableció que al momento de pedir un elemento en un conjunto L bajo el criterio de búsqueda que corresponda (con mínima distancia respecto a un punto o mínima cantidad de pociones requeridas), se retorne con igual probabilidad uno entre los primeros $\alpha \cdot |L|$ elementos ordenados respecto al criterio (uno entre los $\alpha \cdot |L|$ más cercanos cierto punto ó entre los $\alpha \cdot |L|$ que requieren menos pociones). Siendo L en el contexto del problema P o G.

4.4.1. Pseudocódigo

Los cambios se realizaron en las funciones DameMasCercano y DameMasDebil las cuales retornaban un gimnasio o una poke parada para agregar a S.

Algorithm 7 DameMasDebilRandom

```
1: procedure DameMasDebilRandom(L, α)
2:     i ← random % min(|L| · α + 1, 1)
3:     return L_i
4: end procedure
```

Se puede ver facilmente que al implementar S, G y P con listas la complejidad de DameMasDebilRandom es $\mathcal{O}((\alpha+1)\cdot|L|)$, con L parámetro de la función, ya que no se posee acceso aleatorio.

Algorithm 8 DameMasCercanoRandom

```
1: procedure DameMasCercano(L, p, α)
2:     if α = 0 then
3:         res ← BusquedaMasCercanoLineal(L, p)
4:     else
5:         Sort(L)
6:         indice ← RandomNum() % Min(|L| · α + 1, 1)
7:         res ← L_indice
8:     end if
9:     return res
10: end procedure
```

Donde el ordenamiento aplicado a L ordena sus elementos de menor a mayor distancia respecto a p. Se asume esta operación en $\mathcal{O}(|L|\cdot log\,|L|)$ la cual también es la complejidad temporal de DameMasCercanoRandom.

Se concluye que al agregar esta modificación la heurística *MenosPociones* tiene una complejidad temporal $\mathcal{O}(n\cdot log\,n + n\cdot n\cdot\alpha + \hat{p}\cdot m\cdot log\,m) = \mathcal{O}(n\cdot log\,n + n^2\cdot\alpha + \hat{p}\cdot m\cdot log\,m)$.

Se concluye también que la heurística *MasCercano* tiene una complejidad temporal $\mathcal{O}(n^2\cdot log\,n + \cdot m\cdot n\cdot log\,n + m^2\cdot log\,m)$.

Observación 4.1. *Se puede obtener una complejidad $\mathcal{O}(\alpha \cdot |L| \cdot log\,|L|)$ en DameMasCercano-Random utilizando un heap. No se realizó en esta implementación.*

Observación 4.2. *Notar que tomar la implementación anterior es un caso especial de la nueva tomando $\alpha = 0$ y se devuelve siempre el punto más cercano o el gimnasio g con menor $|\text{POC}(g)|$, siendo el único caso determinístico. En caso de tomar $\alpha = 0$, las complejidades originales se siguen cumpliendo.*

4.5. Experimentación

Se implementó un script en python para generar instancias aleatorias del problema en base a un número de poke paradas y gimnasios. Se consideraron tres familias de instancias dependiendo de la densidad de gimnasios en relación a la cantidad de poke paradas con un tope de hasta 2000 nodos $(n + m)$ aumentando con una granularidad de 25:

- proporción $\frac{1}{4} : |G| = \frac{1}{4} \cdot |nodos|$
- proporción $\frac{1}{2} : |G| = \frac{2}{4} \cdot |nodos|$
- proporción $\frac{3}{4} : |G| = \frac{3}{4} \cdot |nodos|$

Para cada instancia evaluada se conservó la distancia obtenida y el tiempo de ejecución. Se buscó obtener resultados determinísticos y replicables por lo que se fijó el valor de α en 0.

4.5.1. *Más Cercano*: optimizando γ

Se buscó aproximar el valor de γ (parámetro del goloso *"Más Cercano"* luego de aplicar la mejora) para el cual se obtienen distancias menores. Se evaluaron las instancias generadas utilizando valores $\gamma \in [0, 1]$ con una granularidad de 0,05. Se normalizaron las distancias obtenidas para instancias con misma cantidad de nodos. Se analizaron los resultados obtenidos por cada *ratio* y también contemplando todos, los resultados se plasmaron en la figura 4.3. Los datos sugieren que los mejores resultados se obtienen utilizando $\gamma \in [0{,}9, 0{,}95]$. Se tomó $\gamma = 0{,}90$.

4.5.2. *Más Cercano* vs *Menos Pociones*: Distancias Obtenidas

Se evaluaron todas las instancias generadas con ambas heurísticas utilizando el γ obtenido en el experimento anterior para *"Más Cercano"*. Los resultados obtenidos en la figura 4.4 sugieren que la heurística *"Más Cercano"* obtiene mejores resultados.

4.5.3. *Más Cercano* Vs *Menos Pociones*: Tiempos de Ejecución

Primero se verificó que efectivamente las complejidades teóricas fueran correctas. Para esto se analizó la correlación entre el tiempo de ejecución obtenido en la experimentación y la complejidad teórica de cada heurísitca ($\mathcal{O}(n^2 + m^2 + n \cdot m)$ *"Más Cercano"* y $\mathcal{O}(n \cdot logn + p \cdot m)$ *"Menos Pociones"*). Se encontró una correlación de 0.98 en ambas heurísticas.

Observación 4.3. *Al analizar la complejidad temporal de los peores casos, el tiempo para una misma cantidad de nodos es el resultado de promediar los valores por sobre el percentil 95.*

Se puede ver como efectivemente la complejidad teórica y el tiempo están fuertemente correlacionados. Observando las figuras 4.5 que corresponden a una regresión lineal entre estas dos variables se puede apreciar como tiempo y complejidad están relacionados linealmente.

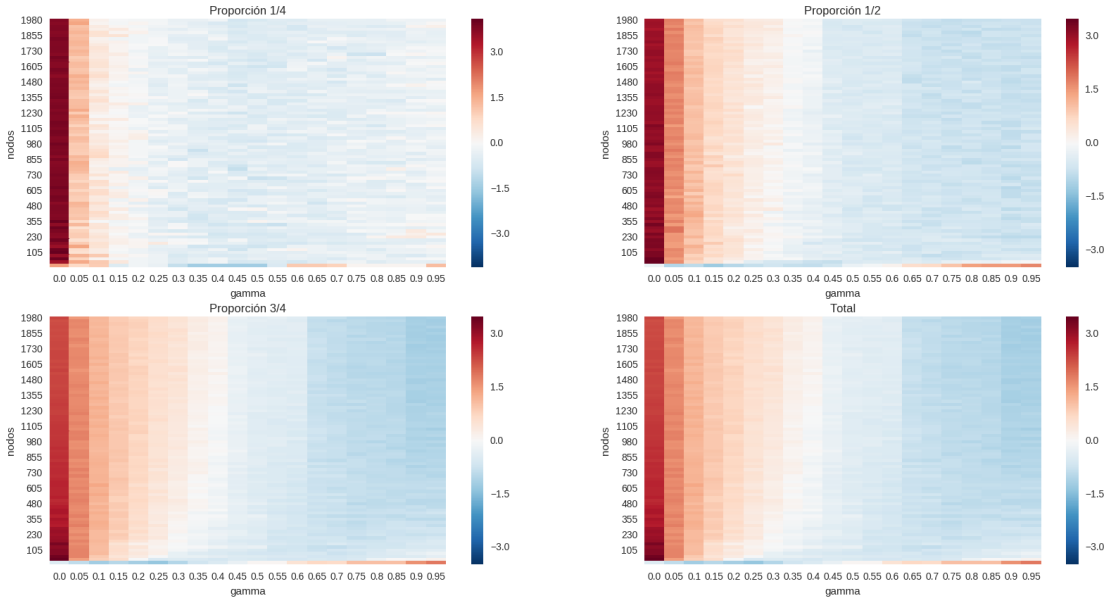


Figura 4.3: Distancias promedio con diferentes γ para diferentes grupos de instancias

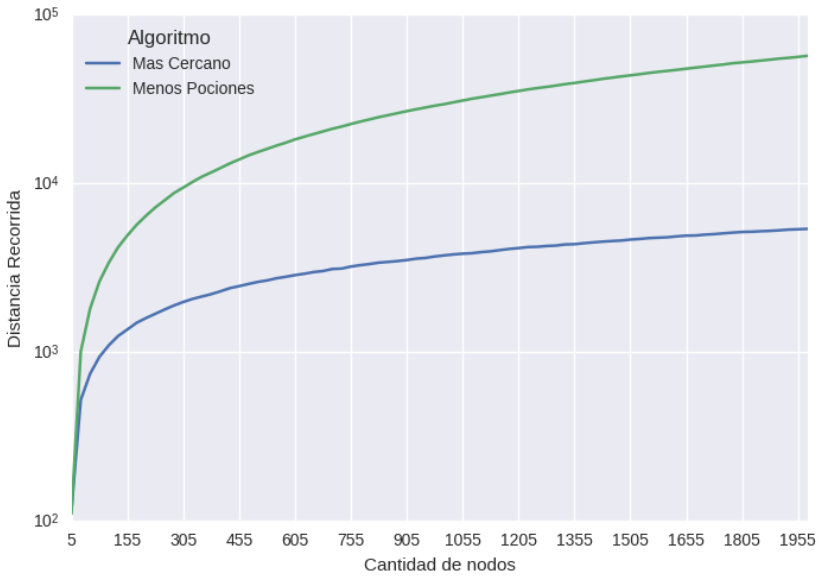
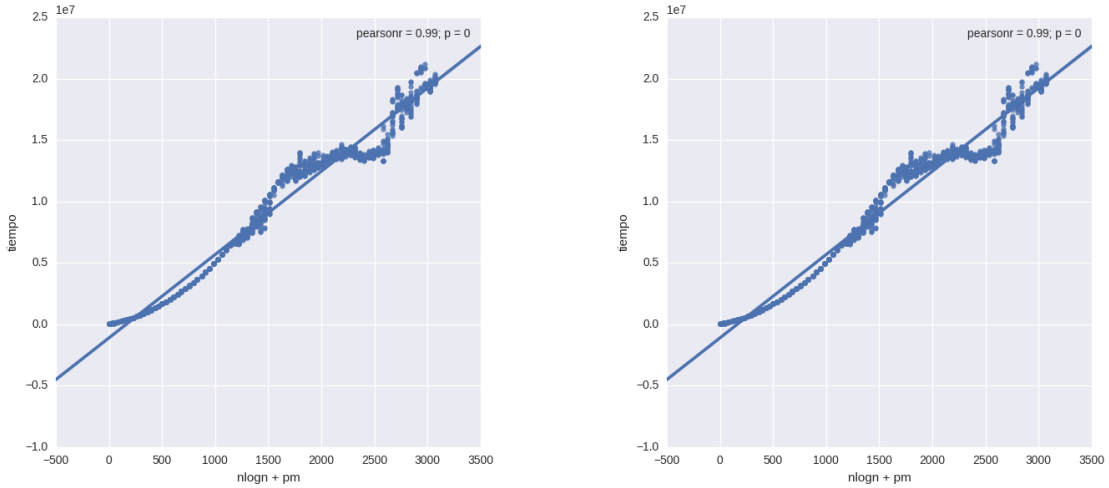


Figura 4.4: *"Más Cercano"* (con $\gamma = 0{,}9$) vs *"Menos Pociones"*



(a) Tiempos de ejecución: *"Más Cercano"*

(b) Tiempos de ejecución: *"Menos Pociones"*

Figura 4.5: Regresión lineal entre tiempo de ejecución y complejidad teórica

Luego se compararon los tiempos de ejecución obtenidos de ambas implementaciones. Se espera habiendo calculado previamente las complejidades temporales de ambas heurísticas, que *"Menos Pociones"* obtenga menores tiempos. Los datos sugieren que las hipótesis se cumplieron.

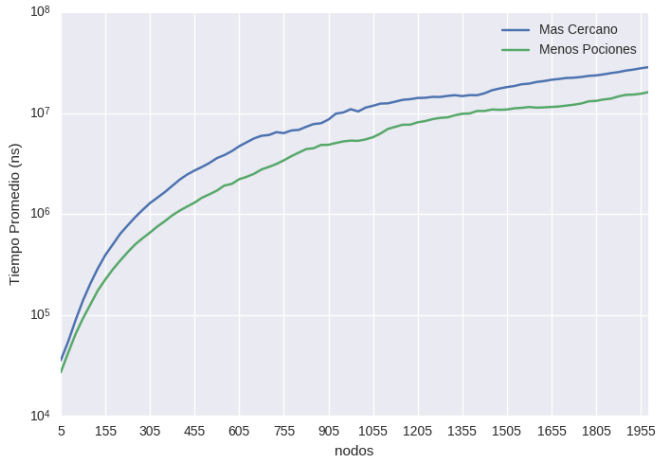


Figura 4.6: Tiempos de ejecución: *"Más Cercano"* vs *"Menos Pociones"*

5. Heurística de búsqueda local

En esta sección se explican las heurísticas de búsqueda local utilizadas. Dada una solución correcta S del problema, la búsqueda local consiste en definir una vecindad de soluciones para S buscando dentro de esta vecindad alguna solución del problema que sea mejor que S. La heurística de búsqueda local realiza sucesivas veces este procedimiento hasta que no haya mejor solución que S en su vecinadrio. En la heurística propuesta la solución inicial se consigue con el algoritmo goloso `mas_cercano` con parámetro de entrada γ 0,9.

Se plantearon dos vecindades, *vecindad lineal* y *vecindad cuadrática*. Ambas se definen de manera similar: se consideran swaps entre nodos de la solución, eliminación paradas e intercambio de paradas dentro de la solución con paradas que no lo estén. La principal diferencia es la dimensión de cada una: la vecindad lineal tiene $\mathcal{O}(n+m)$ vecinos mientras que la otra tiene $\mathcal{O}((n+m)^2)$ vecinos.

Esto define la complejidad de cada uno de los algoritmos. Se busco que en los dos casos cada iteración de la búsqueda tuviera complejidad del orden de la cantidad de vecinos. Para lograr esto se utilizan diversas estructuras auxiliares que otorgan suficiente información para cumplir las complejidades mencionadas.

En base a k_i definido en la sección 2.1 para el nodo s_i, se define $k'_i = k_i$ y p_i como la cantidad de pociones necesarias para que la parte posterior a s_i de la solución sea válida. El valor p_i es una cota inferior de la cantidad de pociones que tendrá que tener la mochila al salir de s_i. De manera análoga a como se define k_i recursivamente, si $p_n = 0$, entonces $p_i = max(p_{i+1} - poc(s_i), 0)$. Valdrá que s_i es válido si $min(k'_i + poc(s_i), k) > p_i$, siendo k la capacidad de la mochila. Dicho menos formal, un nodo será válido si con las pociones que tiene disponibles más las que recibe (o consume), termina con suficientes pociones como para que lo posterior sea válido.

Para las dos vecindades presentadas se utiliza una estructura que captura lo recién explicado para cada nodo de la solución. La figura 5.1 es un caso ilustrativo de esta misma: la primer fila de valores corresponde a los valores $poc(s_i)$, la segunda a los p_i y la tercera a los k'_i.

La idea general del algoritmo de ambas vecindades es, en cada iteración, considerar el costo de cada uno de los vecinos posibles, quedándose con el mejor. Las soluciones vecinas no se generan, sino que se evalúa la mejora de la solución si se realizara dicha operación. Si luego de analizar todas las posibilidades si se encontró algún candidato se hace la corrección necesaria a la solución con la cual se empezó la iteración y se continúa con la búsqueda local. Caso contrario, se finaliza.

5.1. Operaciones de vecindad

En las heurísticas propuestas se utilizan tres operaciones para obtener los vecinos de una solución dada:

- **Eliminación:** Eliminación de una parada dentro de la solución.
- **Swap interno:** Swap entre nodos que se encuentran dentro de la solución.

$$\begin{array}{ccccccc} 3 & & -2 & & 3 & & -4 \\ X & \longrightarrow & X & \longrightarrow & X & \longrightarrow & X \\ 3 & & 1 & & 4 & & 0 \\ 0 & & 3 & & 1 & & 4 \end{array}$$

Figura 5.1: Ejemplo de la estructura auxiliar

- **Swap externo:** Swap entre una parada dentro de la solución y una parada ajena a la solución.

La eliminación funciona de misma manera en ambas vecindades, mientras que los swaps difieren. A continuación se explica cada una de las operaciones y cómo se hace para saber si el swap en cuestión es válido. Considerando que en cada solución vecina el resto del recorrido queda igual, para saber si el vecino es mejor que S (en cuanto a costo total) basta mirar los vecinos de los nodos modificados. Es posible realizar esto en $\mathcal{O}(1)$, pues el grafo que se está utilizando es completo y se conoce la posición eulcídea de cada nodo.

Para saber si estas operaciones son válidas se busca ver evaluar la estructura auxiliar si se hiciera tal modificación. Luego, basta ver que si las posiciones modificadas son válidas.

Para simplificar la notación, q_i denotará $poc(s_i)$.

5.1.1. Eliminación

Para esta primer operación, simplemente se considera como vecino a cada solución que puede obtenerse eliminando alguna parada. De lo mencionado anteriormente, sale que es posible eliminar esta parada si $k'_i \geq p_i$, es decir si no es necesario recargar pociones para continuar el recorrido. Para calcular la ganancia de eliminar esta parada se deben descontar los costos de pasar por la parada, y agregar los costos de avanzar directamente del nodo anterior al siguiente.

5.1.2. Swap interno

Vecindad lineal

En esta vecindad para cada nodo se realizan solamente swaps internos con el nodo contiguo. Para verificar si es válido el swap, basta mirar como quedaría la estructura auxiliar. Un swap interno en el caso siguiente:

$$\begin{array}{ccccc} & q_{j+1} & q_{j+1} & & \\ \rightsquigarrow & X_1 & \longrightarrow X_2 & \rightsquigarrow & \\ & p_j & p_{j+1} & & \\ & k'_j & k'_{j+1} & & \end{array} \quad \text{produce} \quad \begin{array}{ccccc} & q_{j+1} & q_j & & \\ \rightsquigarrow & X_2 & \longrightarrow X_1 & \rightsquigarrow & \\ & max(p_{j+1} - q_1, 0) & p_{j+1} & & \\ & k'_j & min(k'_j + q_2, k) & & \end{array}$$

Lo previo a esta modificación es válido, pues no se han modificado sus pociones. Si se cumple la cota inferior p_{j+1} al salir del nuevo segundo nodo lo posterior también es válido. Entonces, si ambas nuevas posiciones son válidas, el recorrido obtenido es una solución válida. Esto último puede verificarse utilizando el resultado explicado en la introducción de esta sección.

Vecindad cuadrática

En este caso para cada nodo se considera realizar un swap con cualquier otro nodo del recorrido. Sean i y j los índices de los nodos que se desea swapear (con $i < j$). Este swap presenta una dificultad: si bien es fácil saber si lo anterior a s_i y lo posterior a s_j será válido luego del swap, no es directo verificar que los nodos entre s_i y s_j seguirán siendo válidos. Bien podría pasar que al realizar el swap se comience el tramo intermedio con menos pociones, provocando que algún nodo se invalide (en particular puede pasar si $q_i > q_j$ i.e. el nodo j consume más pociones que el nodo i). También podría pasar que al contrario, aumente la cantidad de pociones con la cual se comienza, provocando que se desperdicien pociones y el tramo intermedio sea

inválido (por ejemplo si el nodo j corresponde a una parada y de la posición i ya se salía con la mochila cargada al máximo).

Para sobrepasar esta dificultad se utilizan otras estructuras auxiliares dos matrices cuyos elementos son cotas inferiores o superiores, que dado un intervalo (i, j) (los nodos i y j excluidos) permiten saber si un swap es válido.

La primera recibe el nombre de *minimos*. $minimos[i][j]$ indica la mínima cantidad de pociones disponibles dentro del intervalo (i, j). Por pociones disponibles se entiende la cantidad de pociones que se le puede quitar a un nodo para que siga siendo válido. Este valor permite saber cuántas pociones a lo sumo pueden quitársele al tramo intermedio para que siga siendo válido. Si se le quitan más el nodo correspondiente al mínimo de estos valores no será válido.

La segunda matriz recibe el nombre de *maximos*. $maximos[i][j]$ indica la máxima cantidad de pociones que hubo en la mochila en el intervalo (i, j). Este valor da una cota superior de la cantidad de pociones con las que se puede comenzar el tramo intermedio sin desperdiciar pociones. Si bien el desperdicio de pociones no implica que se invalide la solución, si valdrá que si todo lo otro es válido y no se desperdician pociones, entonces la nueva solución sigue siendo válida.

Con estas dos matrices se puede saber si al realizar el swap se desperdiciarán pociones o si en algún nodo no alcanzarán las pociones. Si no se desperdician pociones y alcanzan para no invalidar los nodos intermedios, basta verificar como quedan los valores p_i y k'_j de los nodos que se swapean. Esto último puede verificarse de manera análoga al caso de la vecindad lineal, permitiendo saber si el recorrido resultante es una solución válida al problema y por lo tanto un vecino a la solución inicial de la iteración de búsqueda local.

Se parte del caso siguiente y se desean swapear los nodos i y j:

$$\begin{array}{ccccc} & q_i & & q_j & \\ \rightsquigarrow & X_i & \rightsquigarrow & X_j & \rightsquigarrow \\ & p_i & & p_j & \\ & k'_i & & k'_j & \end{array}$$

Si los nodos son contiguos se está en la misma situación que la vecindad lineal y no es necesario preopcuprase por el inexistente tramo intermedio. Si no, al alcanzar la posición del nodo i, si se visita X_j en lugar de X_i no se obtienen q_i pociones, si no que se obtienen q_j pociones (recordar que si el nodo es un gimnasio, este valor es negativo). Si se realiza el swap se inicia entonces el tramo intermedio con $dif = q_j - q_i$ pociones. Si $minimos[i][j] + dif \geq 0$ y si $maximos[i][j] + dif \leq k$ [4] se puede asegurar que el tramo intermedio es válido. Falta entonces asegurarse que los dos nodos resultantes del swap sean válidos. La siguiente figura ilustra el resultado de realizar tal swap:

$$\begin{array}{ccccc} & q_j & & q_i & \\ \rightsquigarrow & X_j & \rightsquigarrow & X_i & \rightsquigarrow \\ max(p_j - q_i, 0) & & & p_j & \\ & k'_i & & min(k'_i + q_j, k) & \end{array}$$

Esto vale por asegurar que no se invalida el tramo intermedio. Si se invalidara no se podría asegurar que se llega con $min(k'_i + q_j, k)$ pociones a final del tramo intermedio ni que se necesitan al menos $max(p_j - q_i, 0)$ pociones para comenzar el tramo intermedio.

[4] k es la capacidad de la mochila

Entonces, si el tramo intermedio no se invalida y si los dos nodos resultantes del swap también son válidos implica que el swap sea válido. La mejora obtenida con esta modifación se puede calcular observando que aristas se quitarían y que aristas se agregarían al recorrido.

5.1.3. Swap externo

Una tercer manera de obtener vecinos de una solución S dada, es realizar swaps externos, i.e. considerar hacer un swap entre una parada de S y una parada de las paradas que no se encuentran dentro de la solución. Está operación es la misma para ambas vecindades. La diferencia entre ambas es con cuantas paradas del recorrido se considera realizar un swap externo. La mejora obtenida realizando esta operación se calcula de igual manera a los casos anteriores.

En el caso de la vecindad lineal solamente se hace esta operación para una cantidad acotada de vecinos: en lugar de probar con todas las paradas del recorrido, se prueba con aquellas que hayan sido modificadas en la iteración anterior. En la primer iteración se realiza con las seis primeras paradas que aparezcan en el recorrido.

En la vecindad cuadrática para cada parada del recorrido se prueba swapearla con todas las paradas ajenas al la solución.

5.2. Pseudocódigo y complejidad teórica

Algorithm 9 Pseudocódigo general búsqueda local

```
1: procedure BUSQUEDALOCAL(solucion S)
2:     INICIALIZAR_VALORES(S)
3:     hubo_mejora ← verdadero
4:     while hubo_mejora do
5:         hubo_mejora ← MEJORAR_SOLUCION(S)
6:     end while
7: end procedure
```

Búsqueda local consiste en mejorar lo más posible una solución a lo largo de sucesivas iteraciones (algoritmo 9). La función INICIALIZAR_VALORES se encarga de inicializar la primer estructura explicada. Para inicializar los valores k'_i y p_i alcanza con recorrer la solución primero de inicio a fin y luego de fin a inicio, realizando a lo largo operaciones de costo $\mathcal{O}(1)$, obteniendo un costo de $\mathcal{O}(n+m)$. La función MEJORAR_SOLUCION analiza el vecindario de S para encontrar el vecino que resuelva mejor el problema que se busca resolver y es la función que diferencia ambas vecindades.

5.2.1. Vecindad lineal

El algoritmo 10 corresponde al pseudocódigo de una iteración de mejora de la búsqueda lineal con la vecindad lineal. En la línea 6 se prueba eliminar el nodo si es una parada. En la línea 7 se considera, si el nodo en cuestión es una parada, realizar un swap con alguna paradas externas. En esta vecindad se busca que sea acotada la cantidad de vecinos que se obtienen con esta operación. Para lograr esto, en la primer iteración se marcan arbitrariamente a ser visitadas las primeras seis paradas. En las iteraciones posteriores se hace esto para los nodos que fueron modificados en la iteración anterior, i.e. alguno de sus nodos vecinos o él mismo participaron en la operación previa. En la línea 13 se considera realizar un swap interno. De la explicación en la sección anterior y de los pseudocódigos 11, 12 y 13 resulta evidente que cada una de estas

Algorithm 10 Pseudocódigo mejora de solución (vecindad lineal)

```
1: procedure MEJORAR_SOLUCION_LINEAL(solucion S)
2:     status      ▷ Tiene la información de la iteración (mejor costo, lugar de modificación, ...)
3:     status.MEJORA ← 0
4:     for nodo in S do
5:         if ES_PARADA(nodo) then
6:             CONSIDERAR_ELIMINAR(status, nodo, S)
7:             if FUE_MODIFICADO(nodo) then
8:                 for parada in paradas_afuera do
9:                     CONSIDERAR_SWAP_EXTERNO(status, nodo, parada, S)
10:                end for
11:            end if
12:        end if
13:        if NO_ES_ULTIMO(nodo,S) then
14:            CONSIDERAR_SWAP_INTERNO_LINEAL(status, nodo, S)
15:        end if
16:    end for
17:    if status.HAY_MEJORA_POSIBLE then
18:        APLICAR_MEJORA(status, nodo, S)
19:    end if
20:    INICIALIZAR_VALORES()
21:    return status.HAY_MEJORA_POSIBLE
22: end procedure
```

funciones se realizan en $\mathcal{O}(1)$ operaciones. Las funciones ACTUALIZAR_INFORMACION se encargan de guardar la información necesaria para ser capaz de aplicar efectivamente la mejora a S al final de la iteración. Estas toman $\mathcal{O}(1)$ operaciones.

Algorithm 11

```
1: procedure CONSIDERAR_ELIMINAR(status, nodo n, solucion S)
2:     if ES_POSIBLE_ELIMINAR_PARADA(n, S) then
3:         costo_obtenido ← COSTO_ELIMINANDO(n, S)
4:         if status.COSTO > costo_obtenido then
5:             ACTUALIZAR_INFO_ELIMINACION(status, costo_obtenido, n)
6:         end if
7:     end if
8: end procedure
```

De lo anterior se sigue que el ciclo que comienza en la línea 4 tiene complejidad $\mathcal{O}(n+m)$: cada consideración cuesta $\mathcal{O}(1)$ y se consideran $\mathcal{O}(n+m)$ eliminaciones, $\mathcal{O}(n+m)$ swaps internos y $\mathcal{O}(6 \cdot m)$ swaps externos.

A partir de la línea 17 comienza el final de la iteración. Para finalizarla se debe aplicar la mejora si es que existe. Esto consiste en realizar modificaciones de costo $\mathcal{O}(1)$ a la solución S: eliminar nodos, swapearlos internamete o externamente. Para saber que modifiación se debe hacer se rescata la información que se fue guardando al recorrer los vecinos de S. Se debe además marcar los nodos que fueron modificados (i.e. nodos modificados y sus vecinos), que al ser una cantidad acotada también toma $\mathcal{O}(1)$ operaciones. Finalmente se deben re-inicializar los valores de la estructura auxiliar, que como fue mencionado anteriormente toma $\mathcal{O}(n+m)$.

Algorithm 12

```
1: procedure CONSIDERAR_SWAP_EXTERNO(status, nodo n, nodo parada, solucion S)
2:     costo_obtenido ← COSTO_SWAP_EXTERNO(n, parada, S)
3:     if status.COSTO > costo_obtenido then
4:         ACTUALIZAR_INFO_SWAP_EXTERNO(status, parada, costo_obtenido, n)
5:     end if
6: end procedure
```

Algorithm 13

```
1: procedure CONSIDERAR_SWAP_INTERNO_LINEAL(status, nodo n, solucion S)
2:     if ES_POSIBLE_SWAPEAR_CONTIGUO(n, S) then
3:         costo_obtenido ← COSTO_SWAP_INTERNO_LINEAL(n, S)
4:         if status.COSTO > costo_obtenido then
5:             ACTUALIZAR_INFO_SWAP_INTERNO_CONTIGUO(status, costo_obtenido, n)
6:         end if
7:     end if
8: end procedure
```

Se concluye que cada iteración en la vecindad lineal tiene complejidad $\mathcal{O}(n + m)$.

5.2.2. Vecindad cuadrática

Algorithm 14 Pseudocódigo mejora de solución (vecindad cuadrática)

```
 1: procedure MEJORAR_SOLUCION_CUADRATICO(solucion S)
 2:     INICIALIZAR_MATRICES()
 3:     status      ▷ Tiene la información de la iteración (mejor costo, lugar de modificación, ...)
 4:     status.MEJORA ← 0
 5:     for nodo in S do
 6:         if ES_PARADA(nodo) then
 7:             CONSIDERAR_ELIMINAR(status, nodo, S)
 8:             for parada in paradas_afuera do
 9:                 CONSIDERAR_SWAP_EXTERNO(status, nodo, parada, S)
10:             end for
11:         end if
12:         if NO_ES_ULTIMO(nodo,S) then
13:             for otro_nodo in S from nodo do
14:                 CONSIDERAR_SWAP_INTERNO_CUADRATICO(status, nodo, otro_nodo, S)
15:             end for
16:         end if
17:     end for
18:     if status.HAY_MEJORA_POSIBLE then
19:         APLICAR_MEJORA(status, nodo, S)
20:     end if
21:     INICIALIZAR_VALORES()
22:     return status.HAY_MEJORA_POSIBLE
23: end procedure
```

El algoritmo 14 corresponde al pseudocódigo de una mejora con la vecindad cuadrática. En la línea 2 se inicializan las matrices explicadas anteriormente. Con su respectivo pseudocódigo (16) resulta evidente que esto tiene complejidad $\mathcal{O}((n+m)^2)$ pues consiste en la inicialización de dos matrices de $\mathcal{O}(n+m)$ de lado y dos ciclos anidados para recorrerlas. En la linea 7 se considera la eliminación del nodo si es una parada, lo que toma $\mathcal{O}(1)$ operaciones. A diferencia de la otra vecindad, en este caso con cada parada del recorrido se prueba hacer un swap externo con todas las paradas ajenas a la solución. El ciclo de costo $\mathcal{O}(m)$ en la línea 8 se paga entonces en cada iteración del ciclo principal. En la línea 12 se prueba realizar un swap con todos los nodos posteriores al nodo en cuestión. Cada prueba tiene costo $\mathcal{O}(1)$ (su pseudocódigo 15 es análogo a los ya mencionados), por lo que el ciclo tiene costo $\mathcal{O}(n)$ El cuerpo del ciclo tiene entonces complejidad $\mathcal{O}(n+m)$, por lo que el ciclo principal de la mejora cuadrática tiene complejidad $\mathcal{O}((n+m)^2)$, pues la solución S tiene $\mathcal{O}(n+m)$ nodos.

La finalización de la iteración comienza en la línea 18 y es análoga a la de la vecindad anterior, con la diferencia de que no deben marcarse los nodos que han sido modificados. Esta tiene entonces complejidad $\mathcal{O}(n+m)$.

Resulta entonces que para la vecindad cuadrática cada iteración de la búsqueda local tiene complejidad $\mathcal{O}((n+m)^2))$.

Algorithm 15

```
1: procedure CONSIDERAR_SWAP_INTERNO_CUADRATICO(status, nodo n, nodo otro, solucion
   S)
2:     if ES_POSIBLE_SWAP_INTERNO_CUADRATICO(n, otro, S) then
3:         costo_obtenido ← COSTO_SWAP_INTERNO_CUADRATICO(n, otro, S)
4:         if status.COSTO > costo_obtenido then
5:             ACTUALIZAR_INFO_SWAP_INTERNO_CUADRATICO(status, n, otro, costo_obtenido)
6:         end if
7:     end if
8: end procedure
```

5.3. Experimentación

Los datos obtenidos para esta sección se obtuvieron corriendo las mismas instancias con ambas vecindades. Las instancias tienen cantidad creciente de hasta 1000 nodos y para cada cantidad puntual se generaron instancias distintas que luego se promediaron. En todas las instancias los gimnasios representan la mitad de los nodos, y las paradas la otra mitad. Para cada instancia se guardaron los tiempos que toma cada iteración, en lugar de guardar el tiempo total ya que se busca establecer conclusiones sobre las iteraciones de cada vecindad.

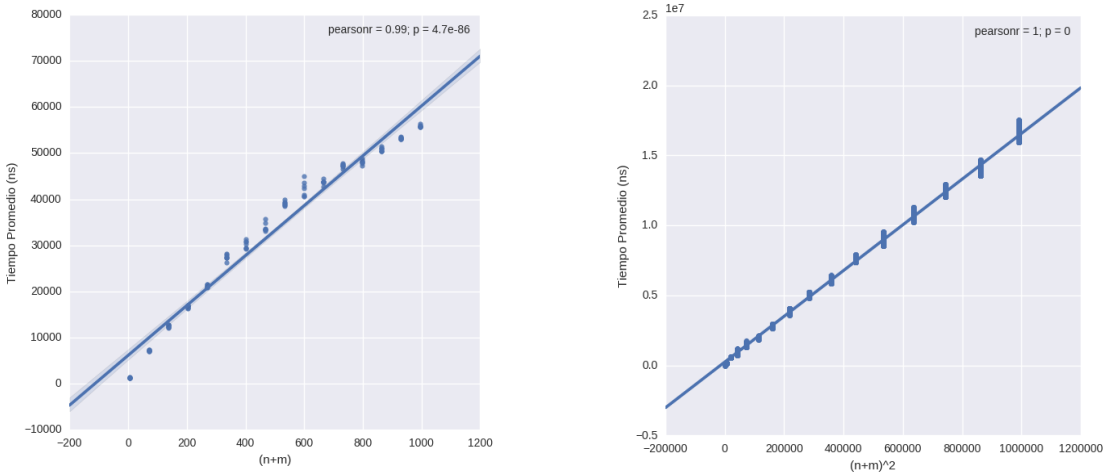
Algorithm 16

```
1: procedure INICIALIZAR_MATRICES(solucion S)
2:     minimos ← matriz de S.SIZE() por S.SIZE()
3:     maximos ← matriz de S.SIZE() por S.SIZE()
4:     for nodo in S until S.ANTEULTIMO  do
5:         i ← nodo.INDEX
6:         max_k ← 0
7:         min_d ← -1
8:         for otro in S from PROXIMO_NODO(nodo,S) do
9:             j ← otro.INDEX
10:            max_k ← MAX(max_k, otro.POCIONES_AL_LLEGAR )
11:            maximos[i][j] ← max_k
12:            q ← otro.POCIONES_QUE_DA
13:            p ← otro.POCIONES_NECESARIAS_AL_SALIR
14:            d ← MIN(p+q, k)- p
15:            if min_d == -1 then
16:                min_d ← d
17:            else
18:                min_d ← MIN(min_d, d)
19:            end if
20:            minimos[i][j] ← min_d
21:            j ← j+1
22:        end for
23:        i ← i+1
24:    end for
25: end procedure
```

5.3.1. Respecto de la complejidad

En primera instancia se busca verificar que efectivamente las complejidades justificadas son correctas. Para esto se considera la correlación entre el tiempo de cada iteración y la complejidad teórica que tendría que tener tal iteración ($\mathcal{O}(n+m)$ para la vecindad lineal y $\mathcal{O}((n+m)^2)$ para la vecindad cuadrática). En el caso de la vecindad lineal se encontró una correlación de 0.985172 entre $(n+m)$ y el tiempo de cómputo. Para la vecindad cuadráticai se encontró una correlación de 0.998276 entre $(n+m)^2$ y el tiempo de cómputo. En ambos casos se ve que la complejidad teórica y el tiempo de corrida están fuertemente correlacionados. La figura 5.2 corresponde a una regresión lineal entre estas dos variables.

Finalmente se realiza una comparación de tiempos entre las iteraciones de cada vecindad en la figura 5.3. Notar que la escala de tiempo es logarítmica.



(a) Vecindad lineal

(b) Vecindad cuadrática

Figura 5.2: Regresión lineal entre tiempo de corrida y complejidad teórica

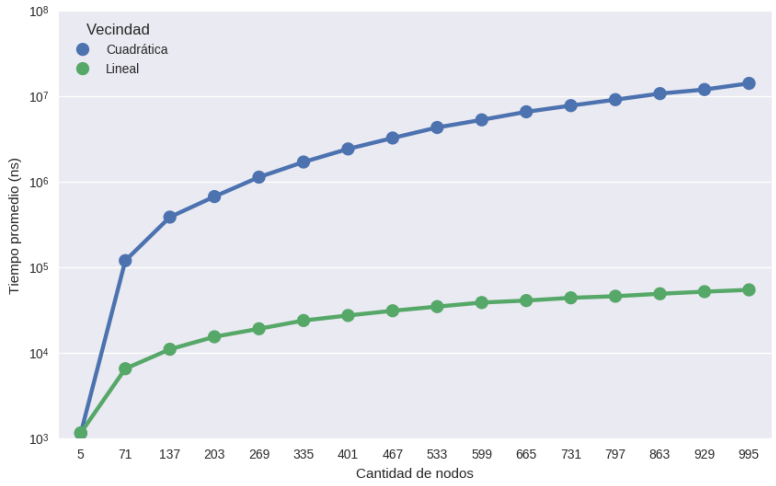


Figura 5.3: Comparación de tiempos de cómputo

5.3.2. Respecto de la cantidad de iteraciones

Por otro lado, se busca establecer una aproximación para la cantidad de iteraciones que se realizan al aplicar la búsqueda local para cada vecindad en función del tamaño de la entrada. Esto no pudo hacerse de manera teórica dado que no hay en el algoritmo ningún indicio que indique cuántas iteraciones serán necesarias hasta no encontrar nada mejor entre los vecinos. Lo única hipótesis que se tiene es que la vecindad cuadrática realizará más iteraciones que la vecindad lineal. Esto se debe a que la primera explora más vecinos que la segunda, aumentando así las posibilidades de que encontrar un vecino que tenga mejor distancia que la de la solución que se tiene en el momento.

Se busca entonces establecer de manera empírica la relación entre la cantidad de iteraciones y cada vecindad. Con instancias de tamaño creciente se estudió la cantidad de iteraciones que tomaba cada vecindad. Entre la cantidad de iteraciones y el tamaño de la entrada (cantidad de nodos) se encontró una correlación de 0.967 para la vecindad lineal y de 0.969 para la vecindad cuadrática.

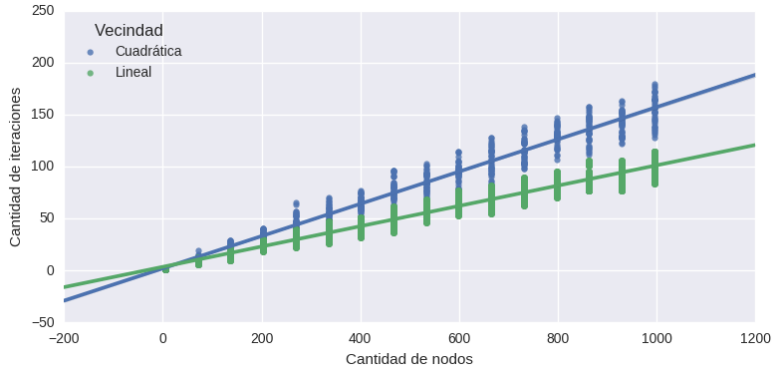


Figura 5.4: Regresión lineal entre la cantidad de iteraciones y la cantida de nodos

El gráfico en la figura 5.4 muestra dos regresiones lineales (una para cada vecindad) entre la cantidad de nodos y la cantidad de iteraciones que toma aplicar la búsqueda. Evidentemente la relación entre ambas es lineal.

Este resultado permite inferir dos conclusiones. Por un lado se puede establecer cierta aproximación para la cantidad de iteraciones necesarias para cada vecindad. Observando la figura recién mencionada y las respectivas rectas obtenidas con la regresión lineal, resulta que la recta para la vecindad lineal tiene una pendiente de aproximadamente $\frac{10}{100}$ mientras que la correspondiente a la cuadrática de $\frac{16}{100}$. De esto se puede suponer que la búsqueda local con la vecindad lineal toma aproximadamente $\frac{10}{100} \cdot (n+m)$ iteraciones, mientras que la cuadrádtica toma aproximadamente $\frac{16}{100} \cdot (n+m)$.

Por otro lado, también vale que la cantidad de iteraciones para una entrada de tamaño $n+m$ es $\mathcal{O}(n+m)$. Esto nos permite establecer cotas de complejidad para la totalidad de las búsquedas locales para ambas vecindades: su complejidad es la cantidad de iteraciones por el costo de cada iteración.

5.3.3. Respecto de los resultados

Con los resultados anteriores resulta evidente que realizar búsqueda local con la vecindad lineal es más rápido que con la vecindad cudrática, lo cual es esperable pues la segunda explora

más candidatos. Esto último también se muestra viendo la cantidad de iteraciones de cada una (cada iteración es un candidato más al cual se le exploran los vecinos).

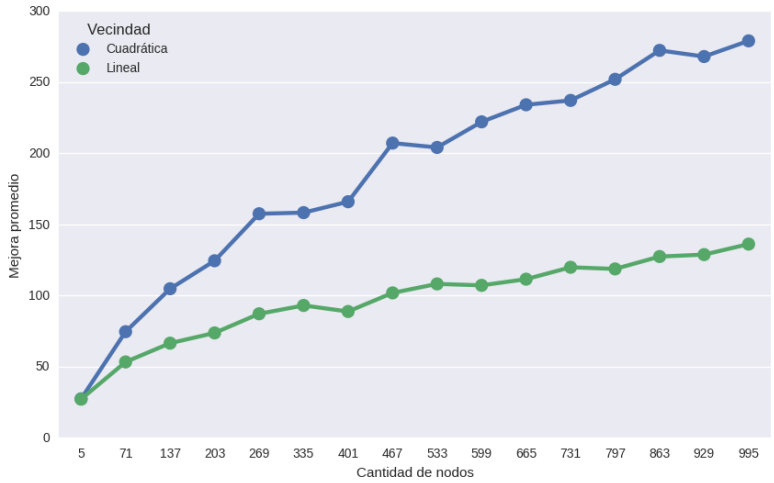


Figura 5.5: Comparación de las mejoras obtenidas con cada vecindad

Sin embargo también es relevante la calidad de los resultados obtenidos con cada vecindad. La figura 5.5 ilustra las mejoras obtenidas con cada vecindad. Resulta que las mejoras respecto de la solución inicial que se obtienen con la vecindad cuadrática son aproximadamente el doble de lo que se obtiene con la vecindad cuadrática. Es evidente entonces que si se dispone del tiempo suficiente, conviene utilizar la vecindad cuadrática pues se obtiene un mejor resultado.

Sin embargo, en el contexto de este trabajo práctico esta búsqueda local se utilizará repetidamente dentro de la metaheurística GRASP. No queda claro si convendrá utilizar una heurística de búsqueda local más rápida o más exhaustiva. Si bien da peores resultados, dado que la vecindad lineal está en un orden estricto menor de complejidad se plantea la hipótesis de que será mejor para GRASP que la vecindad cuadrática pues permitirá más iteraciones en menos tiempo.

6. Meta-Heurística GRASP

La meta-heurística GRASP consta en utilizar un algoritmo goloso random, es decir que tenga un método de selección random, y luego buscar mejorar el resultado obtenido con un algoritmo de búsqueda local. Se realiza este proceso de refinamiento una cantidad de veces definida por una o varias condiciónes de corte.

Se plantearon los siguientes cortes:

- SIMPLE: corta luego de una determinada cantidad de iteraciones.
- RESETEADOR: renueva la cantidad de iteraciones al encontrar una mejor solución.

Algorithm 17 Pseudocódigo de GRASP Reseteador

```
1:  procedure DO_GRASPTILLDEAD(greedy, local, iter, iter_total, α, γ)
2:      j ← 0
3:      i ← 0
4:      while i < iter ∧ j < iter_total do
5:          if greedy then
6:              golosoRes ← GREEDY_MAS_CERCANO(α, γ)
7:          else
8:              golosoRes ← GREEDY_MENOS_POCIONES(α)
9:          end if
10:         if goloso consiguió solución then
11:             if primera vez que entra en el ciclo then
12:                 min ← golosoRes
13:             end if
14:             if local then
15:                 localRes ← APLICAR_BUSQUEDA_LOCAL_LINEAL
16:             else
17:                 localRes ← APLICAR_BUSQUEDA_LOCAL_CUADRATICA
18:             end if
19:             if localRes < min then
20:                 min ← localRes
21:                 i ← 0
22:             end if
23:         end if
24:         j ← j+1
25:         i ← i+1
26:     end while
        return min
27: end procedure
```

Donde

- GREEDY y LOCAL indican la implementación de goloso y búsqueda local a utilizar.
- ITER y ITER_TOTAL son las condiciones de corte *Reseteador* y *Simple* respectivamente.
- α y γ son utilizados por las heurísticas golosas[5].

[5] Ver página 9

6.1. Experimentación

6.1.1. Iteraciones

Se realizaron varias ejecuciones del algoritmo para diferentes instancias. Se varió la cantidad de nodos (gimnasios y poke paradas) desde 5 a 500 con una granularidad de 25 y un ratio entre gimnasios y poke paradas de $\frac{1}{2}$, generando 50 instancias aleatorias distintas para cada una. Se fijó la utilización del goloso *Más Cercano* con $\gamma = 0{,}9$, parámetro para el cual los datos mostraron obtener mejores resultados. Se tomó $\alpha = 0{,}25$ para realizar elecciones aleatorias pero no desmedidas. Como condición de corte se utilizó *Reseteador*, de 10 a 200 en el caso de la búsqueda local lineal con saltos de a 10 y de 2 a 20 saltando de a 2 en cuadrático. Se decidió también no utilizar el corte *Simple* para evaluar el comportamiento de la heurística. En cada ejecución se conservó:

- ITERACIONES FIJAS: cantidad de iteraciones iniciales que setea el corte *Reseteador*.
- ITERACIONES REALIZADAS: cantidad de iteraciones que termina realizando el corte *Reseteador* en total.

Dado que las mejoras de las distancias son estrictas y la distancia mínima es acotada, se espera encontrar un valor de ITERACIONES FIJAS a partir del cual las mejoras no fueran significativas. Los resultados se representaron en las figuras 6.1 y 6.2. El rango de los colores del *heatmap* representa la distancia estandarizada.

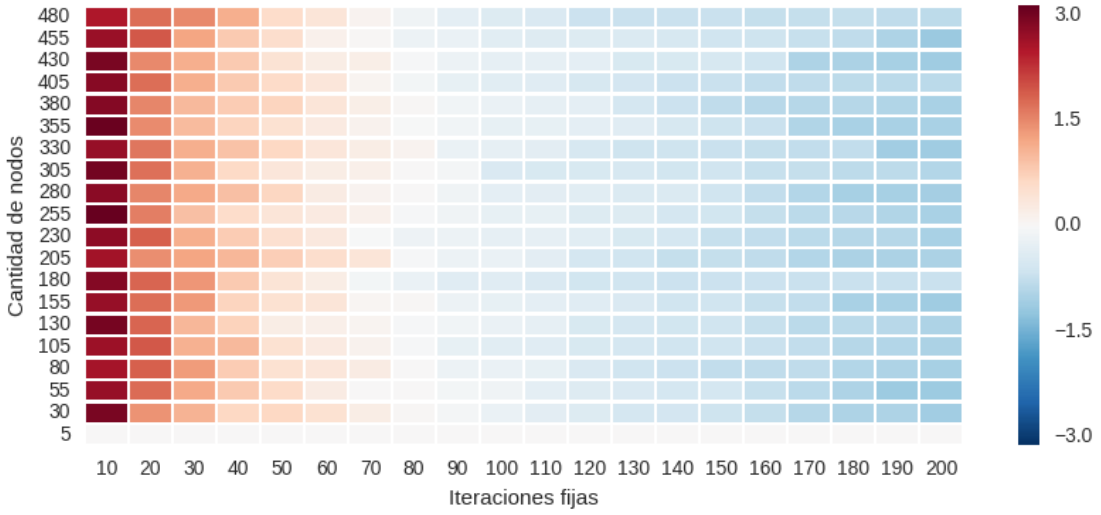


Figura 6.1: GRASP corte Reseteador: Local Lineal

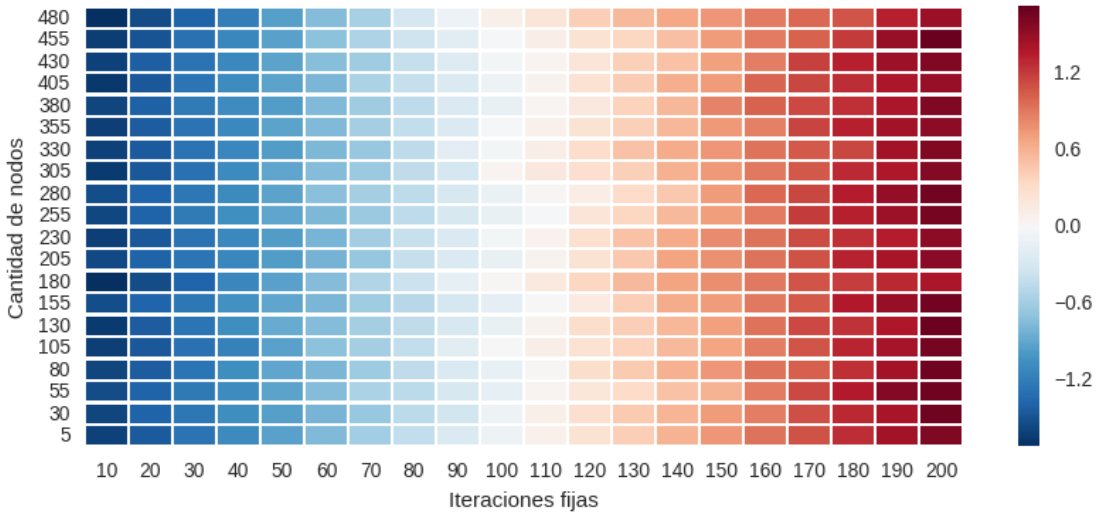


Figura 6.2: GRASP corte Reseteador: Local Cuadrático

Los datos sugieren que al aumentar la cantidad de ITERACIONES FIJAS la distancia diminuye. Se puede apreciar esto también promediando las distancias estandarizadas de cada columna como en las figuras 6.3 y 6.4 .

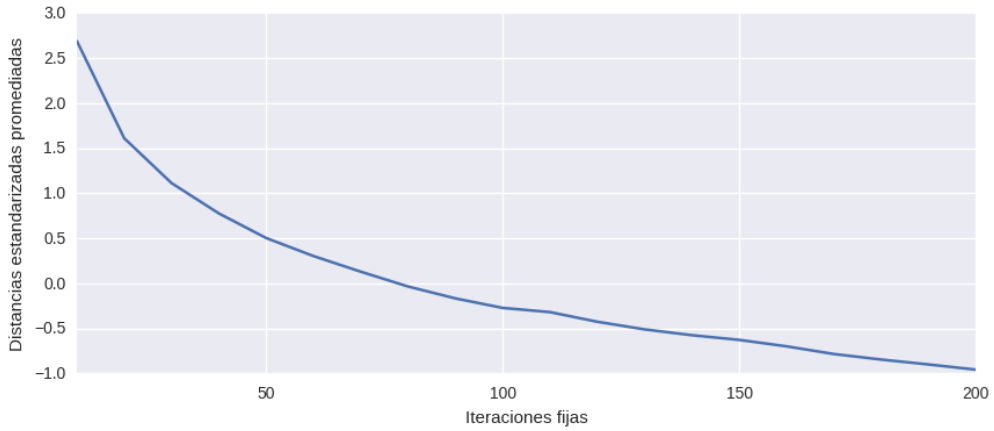


Figura 6.3: Distancias estandarizadas promediadas Lineal

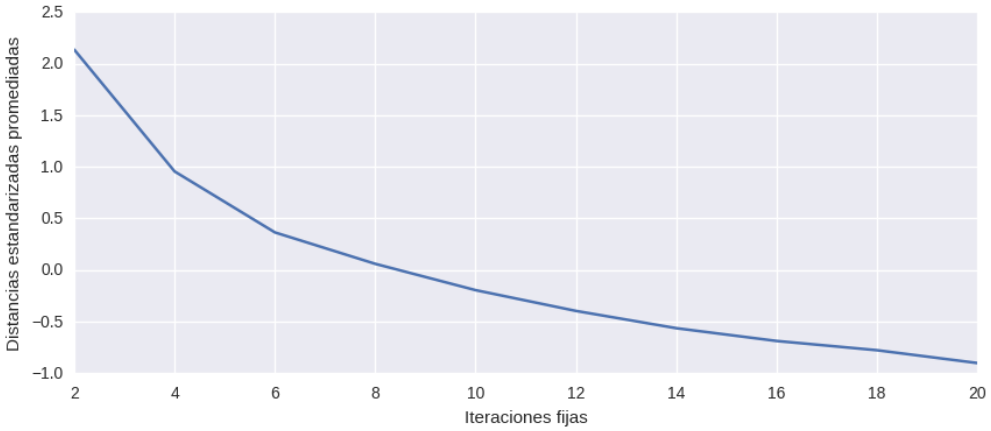


Figura 6.4: Distancias estandarizadas promediadas Cuadrático

Los resultados obtenidos indican que al aumentar la cantidad de iteraciones disminuyen las distancias y también las mejoras obtenidas.

Si bien no se obtuvieron los resultados asintóticos esperados, es decir, no se encontró un valor de ITERACIONES FIJAS para el cual la mejora obtenida se mantenga relativamente constante, no quiere decir que esto no suceda para valores más grandes. Teniendo en cuenta el tiempo de cómputo requerido para ambas implementaciones y los resultados de distancias (se pueden ver en la sección 7) se decidió utilizar como ITERACIONES FIJAS 10 y 200 para la búsqueda cuadrática y lineal respectivamente.

6.1.2. Nuevo Corte

Dado que utilizando sólo la condición de corte *Reseteador* el algoritmo puede ejecutarse indefinidamente, se necesita establecer una cantidad de iteraciones fija para el corte *Simple* de manera tal que afecte lo menos posible la condición *Reseteador* requerida.

Se analizó la relación entre ITERACIONES FIJAS e ITERACIONES REALIZADAS en base a los datos obtenidos en el experimento anterior. Los resultados se pueden ver en las figuras 6.5 y 6.6.

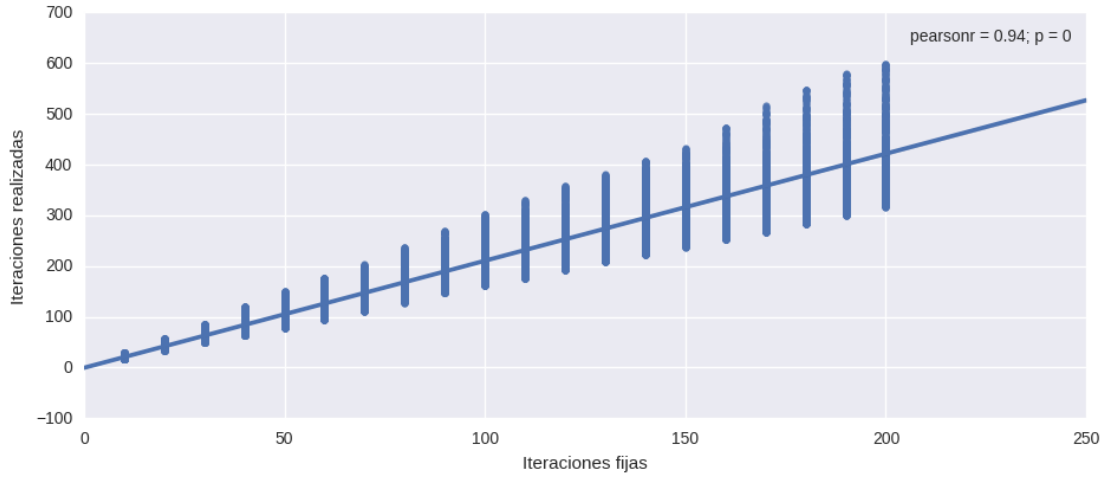


Figura 6.5: GRASP corte Reseteador Lineal: Iteraciones realizadas vs Iteraciones fijas

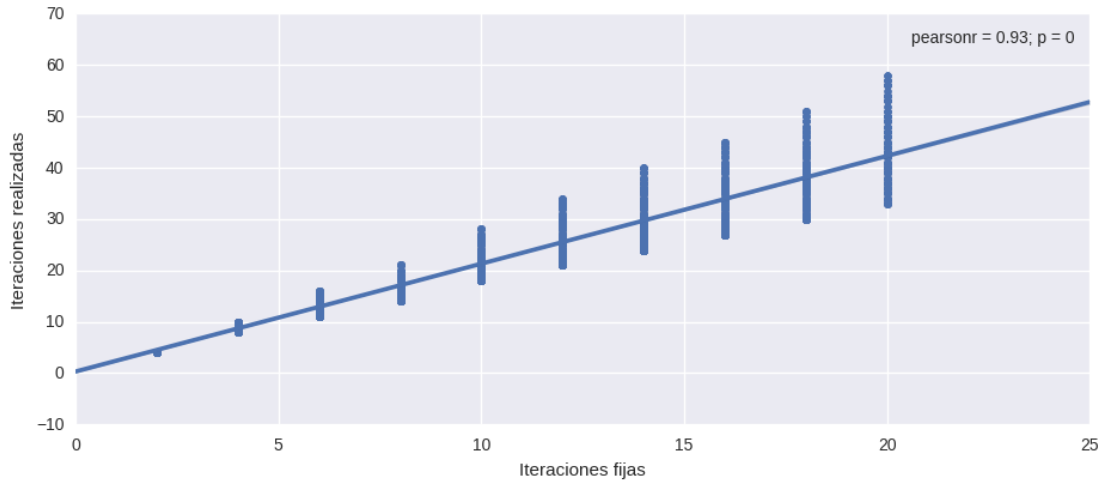


Figura 6.6: GRASP corte Reseteador Cuadratico: Iteraciones realizadas vs Iteraciones fijas

Los datos sugieren la existencia de una correlación lineal entre ambos que permite calcular la cantidad de iteraciones totales en base a las ITERACIONES FIJAS requeridas. Se optó por tomar la constante 4 agregando al algoritmo un corte *Simple* de 4· ITERACIONES FIJAS.

6.1.3. α óptimo

Se utilizaron las mismas instancias descritas en el experimento anterior fijando ITERACIONES FIJAS en 200 para la búsqueda local y 10 para la cuadrática y tomando para α los valores 0,01, 0,03, y desde 0,05 a 0,95 con una granularidad de 0,05 en el caso de la búsqueda lineal, y con un tope de 0,65 en el caso de la búsqueda cuadrática. Las figuras 6.7 y 6.8 muestran las distancias obtenidas para las distintos valores utilizados.

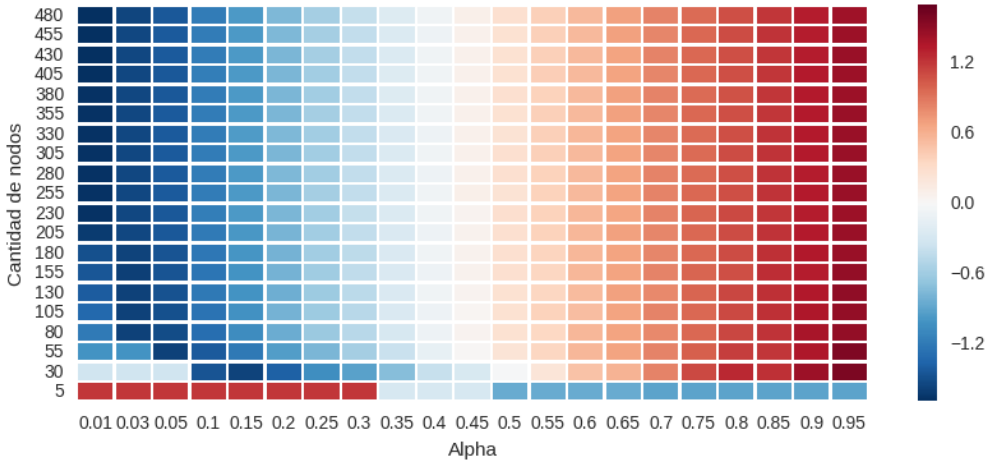


Figura 6.7: GRASP corte Reseteador Lineal: α x Nodos

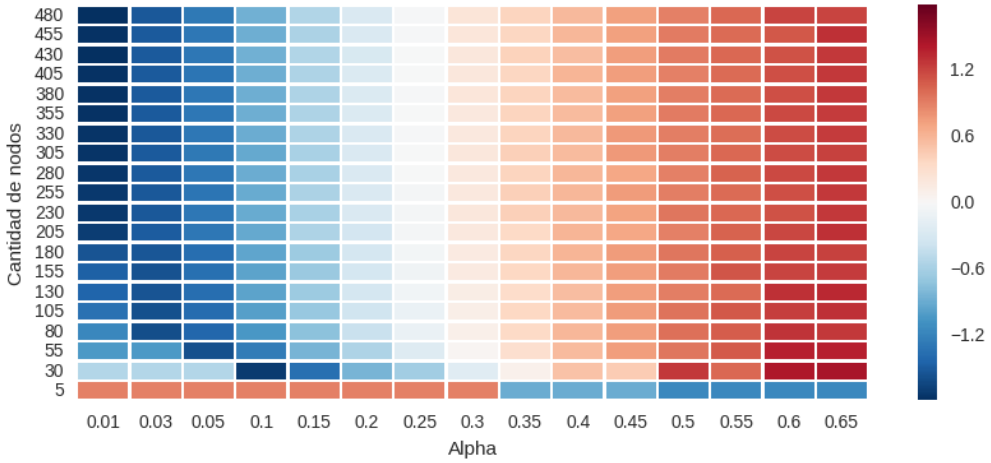


Figura 6.8: GRASP corte Reseteador Cuadratico: α x Nodos

Los datos sugieren que mientras más chicos sean los valores de α mejores distancias se obtienen (más chicas). Se puede ver una anomalía en los resultados obtenidos con instancias menores a 205 nodos. No se logró encontrar una justificación y se enmarca en el contexto de estudios pendientes.

7. Resultados finales y conclusiones

Se realizó un nuevo set de experimentos utilizando 100 instancias diferentes a las utilizadas anteriormente con el objetivo de realizar una comparación final entre todas las heurísticas implementadas. Todas las instancias fueron generadas aleatoriamente y tienen la misma cantidad de gimnasios y poke paradas. Como parámetros de grasp se utilizo un α de 0.01 y 100 iteraciones fijas para el que utiliza la vecindad lineal y 10 para el que utiliza la cuadrática.

Se separaron instancias de tamaños chicos para realizar una comparación con el algoritmo exacto, si bien para algunas instancias se alcanza el minimo (ver figura 3.1), solo ocurre cuando la cantidad de nodos es baja.

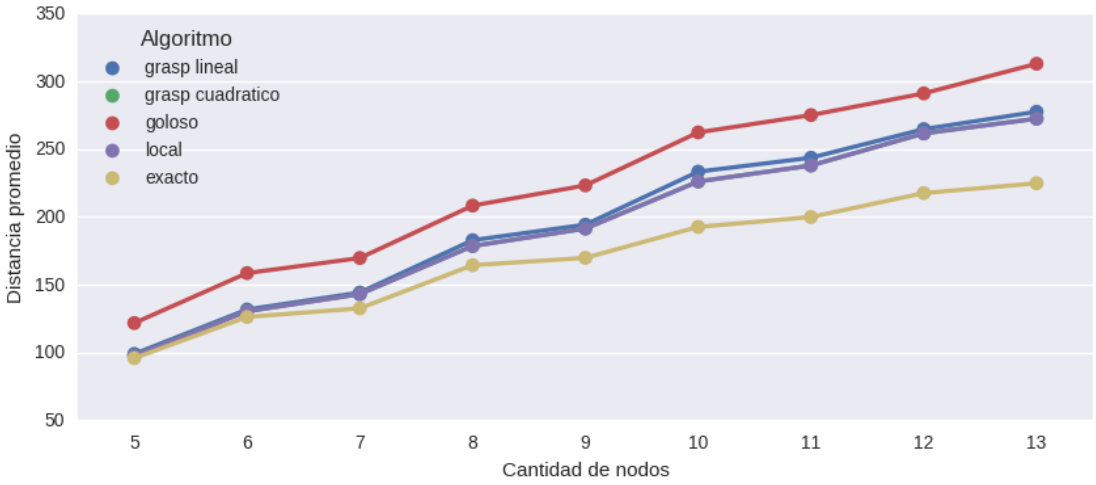


Figura 7.1: Distancia promedio obtenida por los distintos algoritmos

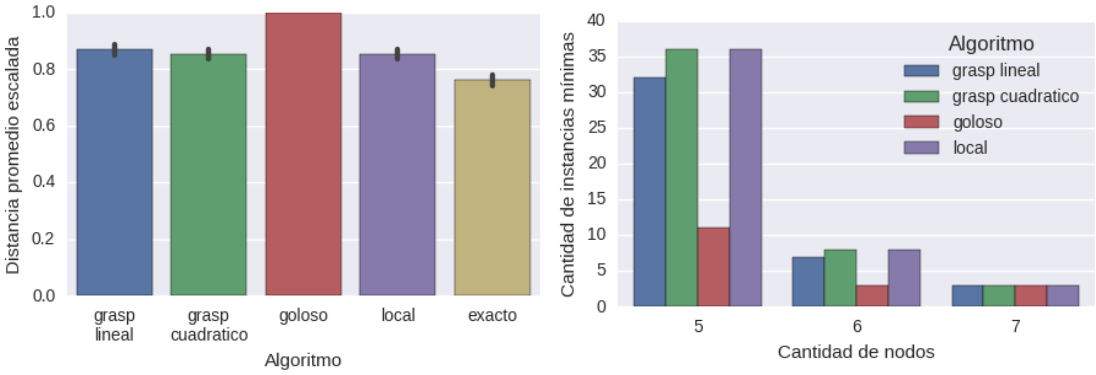


Figura 7.2: Relacion de distancias entre los distintos algoritmos y cantidad de instancias (de 100) que lograron un mínimo

Para las instancias de tamaños mayores (ver figura 7.3) se puede ver que ambas implementaciones de grasp mantienen promedios similares en cuanto a la distancia obtenida. Para el rango de nodos entre 200 y 400 obtiene mejores resultados aunque se observa que la tendencia es a

subir incluso por encima de la busqueda local. Una hipótesis posible es que una mayor cantidad de nodos requiere un valor α aún más chico. Así como las instancias de menos de 200 nodos parecen requerir un valor α mayor (este resultado se puede ver en 6.1.3).

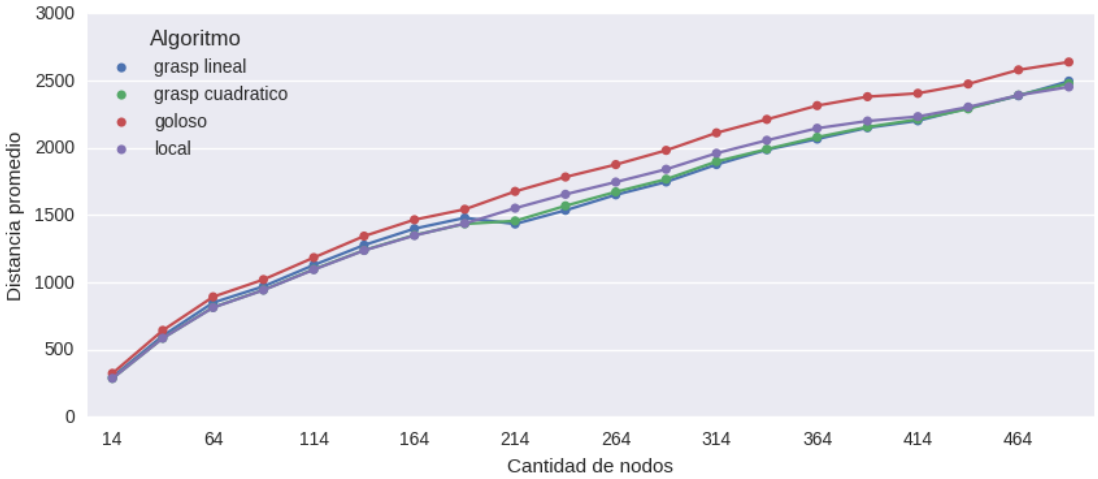


Figura 7.3: Distancia promedio obtenida por los distintos algoritmos

7.1. Análisis de tiempos

En la figura 7.5 se puede ver que el tiempo de ejecucion que requiere el cuadrático en relacion al lineal es aproximadamente el doble a los 500 nodos y tiende a una diferencia mayor. Dado que ambos generaron distancias similares, grasp lineal parece obtener mejores resultados. El costo computacional de grasp en relación a la busqueda local es importante y los resultados de distancias con instancias de tamaños grandes casi no muestran diferencia, o la misma favorece al local.

Si bien se mostró que por lo general la busqueda local lineal es peor que la cuadrática, grasp parece obtener mejores resultados cuando cada iteración es rápida, esto permite obtener y explorar mas soluciones aleatorias en menos tiempo.

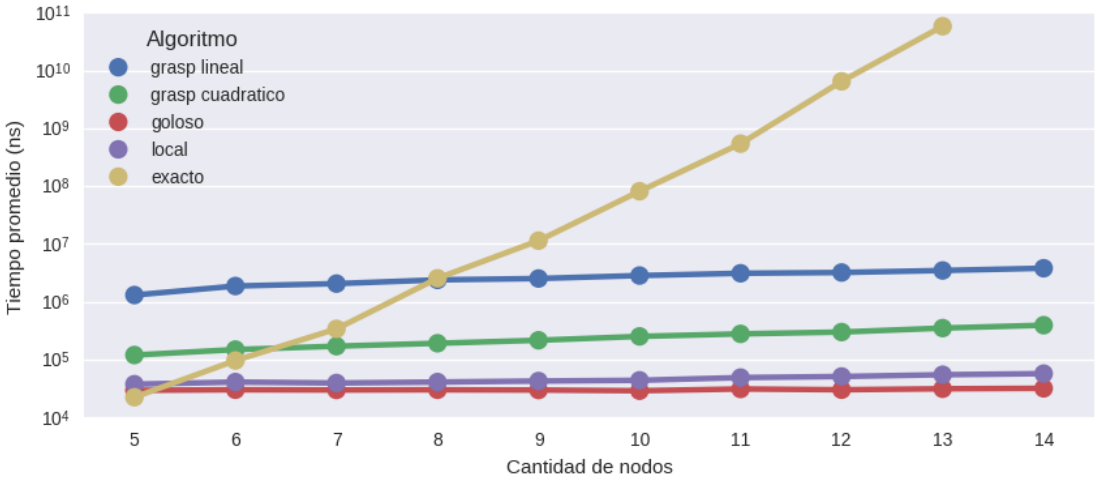


Figura 7.4: Tiempo de cómputo de los distintos algoritmos

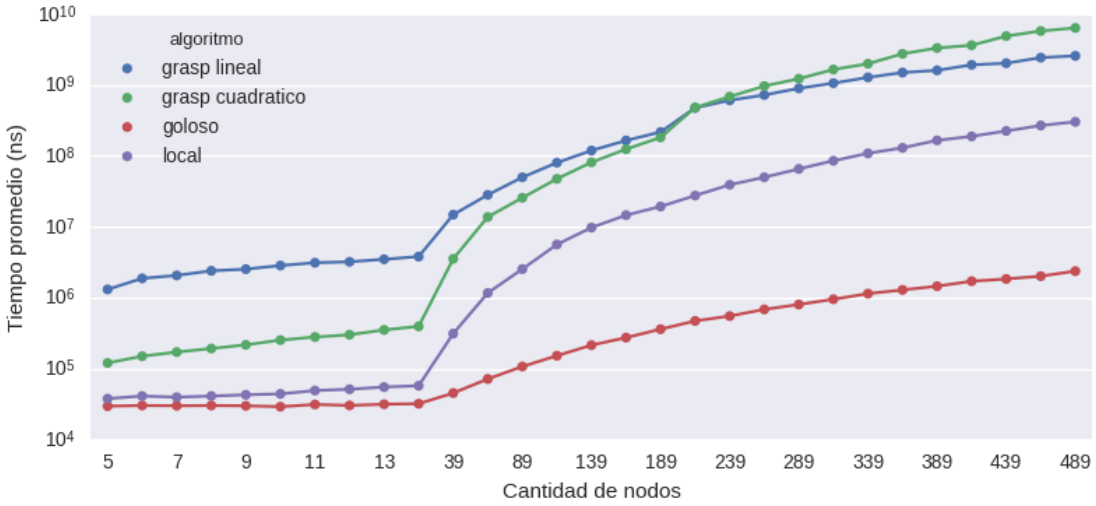


Figura 7.5: Tiempo de cómputo de los distintos algoritmos

8. Otras posibles alternativas experimentales

Algunas variables y consideraciones no fueron investigadas y pueden proveer resultados interesantes, entre ellas se destacan las siguientes:

- Distintos valores k (tamaño de mochila): Un mayor valor de k puede permitir obtener un mejor tiempo para una misma instancia. El comportamiento de las diferentes heurísticas puede verse modificado. Una hipótesis es que greedy mas cercano con desvíos puede obtener mejores resultados. También se puede estudiar el comportamiento de las heurísticas para instancias que no cumplan la conficion suficiente.
- Diferentes ratios: Evaluar el comportamiento de los algoritmos con diferentes relaciones de gimnasios y poke paradas.
- Experimentación más profunda de grasp: Se ve en la figura 7.3 que el comportamiento de grasp a medida que el tamaño de las instancias incrementa empeora en relacion a la busqueda local. Puede ser interesante evaluar los resultados con instancias de tamaños mayores y diferentes valores de α. También evaluar el desempeño del algoritmo con la busqueda golosa *Menos Pociones*, tal vés consigue un mejor espacio de soluciones y tiene una interacción con la busqueda local diferente.
- Diferentes cotas para exacto: Una solución de busqueda local se puede utilizar como cota para implementar podas en el algoritmo exacto, las mismas pueden permitir la resolución de instancias de tamaños mayores. También se puede ajustar la poda utilizando un árbol generador mínimo (AGM) con los gimnasios restantes dado que un camino hamiltoniano entre los gimnasios es mayor al peso del AGM. Si bien se obtiene una mejor cota, puede resultar que el tiempo requerido para la generación del árbol empeore el desempeño del algoritmo.